Parsing Wrap-Up
Thoughts on Assignment 4
PL Feature: Type System

CS 331 Programming Languages

Lecture Slides

Wednesday, February 18, 2026

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Unit Overview
Lexing & Parsing

Topics

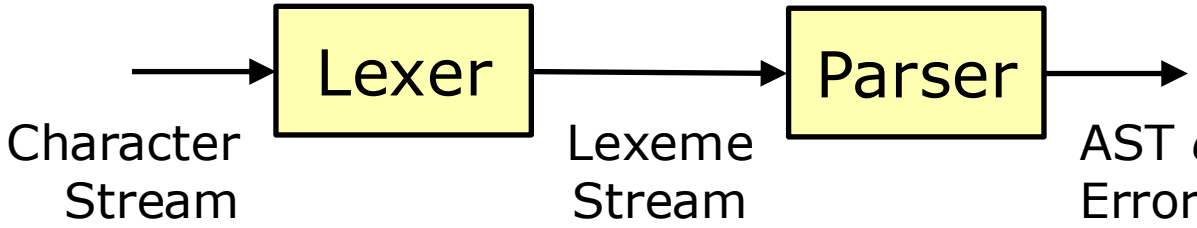
- ✓ Introduction to lexing & parsing
- ✓ The basics of lexical analysis
- ✓ State-machine lexing I
- ✓ State-machine lexing II
- ✓ State-machine lexing III
- ✓ The basics of syntax analysis
- ✓ Recursive-descent parsing I
- ✓ Recursive-descent parsing II
- ✓ Recursive-descent parsing III
- ✓ Shift-reduce parsing
- Parsing wrap-up

Lexical Analysis (Lexing)

Syntax Analysis (Parsing)

Review

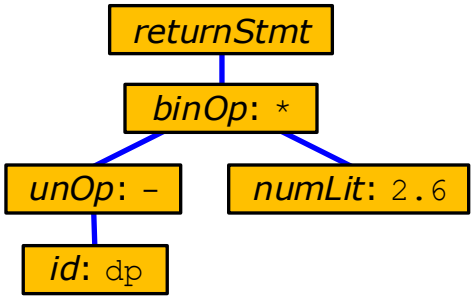
Review
Introduction to Lexing & Parsing




```
return (-dp * 2.6) // x
```

```
return (-dp * 2.6) // x
```

key, op, id, op, num lit, punct, punct



Two steps:

- **Lexical analysis** (**lexing**)
- **Syntax analysis** (**parsing**)

The output of a parser is typically an *abstract syntax tree* (*AST*). Specifications of these vary.

The Basics of Syntax Analysis

Parsing methods can be divided into two broad categories: *top-down* and *bottom-up*.

Top-Down Parsers

- Go through derivation top to bottom, **expanding** nonterminals.
- Sometimes hand-coded and sometimes automatically generated.
- Method we look at: **Predictive Recursive Descent**.

Bottom-Up Parsers

- Go through the derivation bottom to top, **reducing** substrings to nonterminals.
- Code is almost always automatically generated.
- Method we look at: **Shift-Reduce**.

Review
Shift-Reduce Parsing [1/3]

Shift-Reduce parsing is a table-based bottom-up method.

Parser execution uses a state machine with a stack, called a **Shift-Reduce automaton**.

Each stack item holds a symbol and a state.

Stack

expr	8
=	4
ID (a)	2
	1

Operation is specified by a **parsing table** in two parts—**action table** & **goto table**—constructed before execution.

At each time step a lookup in the action table is done. One of four operations will be specified.

- **SHIFT.** Shift the next input symbol onto the stack.
- **REDUCE.** Apply a production in reverse, reducing symbols on the stack to a single nonterminal. See the goto table for the new state.
- **ACCEPT.** Done, syntactically correct.
- **ERROR.** Done, syntax error.

Review
Shift-Reduce Parsing [2/3]

If a Shift-Reduce parser does not do multi-symbol look-ahead (which is typical), then the grammars a correct parser can be based on are called the **LR(1) grammars**.

The natural grammar for expressions with left-associative binary operators is typically an LR(1) grammar. Unlike Predictive Recursive Descent, Shift-Reduce parsing has no problem with productions that involve left recursion, like the following.

expr → *term*
| *expr* ("+" | "–") *term*

Review
Shift-Reduce Parsing [3/3]

A description of a parsing method includes both how the Shift-Reduce parser works and how the parsing tables are generated.

Generating Shift-Reduce parsing tables in a straightforward way tends to result in large, impractical tables. Various ways of generating smaller tables are known, but these generally do not work for all LR(1) grammars.

Probably the most common way of generating smaller parsing tables involves a procedure for merging states [F. DeRemer 1969]. The LR(1) grammars for which this gives correct results are the **LALR(1) grammars** (**Look-Ahead LR(1) grammars**).

The **Yacc** parser generator and its descendants (GNU **Bison**, etc.), generally default to generating parsing tables using this method.

Parsing Wrap-Up

Parsing Wrap-Up
Efficiency of Parsing [1/5]

We have discussed parsing algorithms that can handle some, but not all, CFLs. We have not yet discussed their efficiency.

How can we analyze the running time of a parsing (or lexing) method?

When we analyze algorithms, we need to be clear on three things:

- How we measure the size of the input (usually denoted by *n*).
- What operations are allowed.
- What operations we count (the **basic operations**).

Parsing Wrap-Up
Efficiency of Parsing [2/5]

One reasonable option:

- The **size of the input** (n) is the number of characters in it.
- **Allowed operations** are reading a single input character, along with any internal processing operations we want.
- **Basic operations** are reading a single input character, along with the usual C operators.

This works well when the input is a character stream—analyzing:

- A lexer.
- A parser in which lexing is not a separate step.
- A lexer & parser considered together, as a single processing step.

For a parser considered in isolation, with the lexer being separate, we can let n be the **number of *lexemes* in the parser's input**. Revise the basic operations to include lexeme operations: reading a lexeme, copying a lexeme, comparing two lexemes.

Parsing Wrap-Up
Efficiency of Parsing [3/5]

In all cases, *for a fixed grammar or lexeme description*, each of the methods we have discussed (State-Machine lexing, Predictive Recursive-Descent parsing, Shift-Reduce parsing) is linear time.

Example 1. For a State-Machine lexer, a state cannot be repeated without advancing the input (otherwise we have an infinite loop), and there are a fixed number of states.

Example 2. For a Predictive Recursive-Descent parser, a parsing function cannot be called multiple times without advancing the input, and there are a fixed number of parsing functions.

This turns out to be true in general for practical lexers and parsers.

Practical lexers and parsers run in linear time.

Parsing Wrap-Up
Efficiency of Parsing [4/5]

Recall that the parsing methods we covered cannot handle all CFLs.

In the late 1960s and 1970s, parsing methods were found that can handle all CFLs.

- **Earley Parser** [J. Earley 1968]
- **CYK Parser** [J. Cocke & J.T. Schwartz 1970, T. Kasami 1965, D.H. Younger 1967]
- **Generalized LR** (**GLR**) **Parser** [B. Lang 1974, M. Tomita 1984]

Note!

All of these have a worst-case time of $\Theta(n^3)$ for an arbitrary CFL. Some (Earley and GLR, in particular) are faster for many CFGs, and linear-time for some.

Parsing Wrap-Up
Efficiency of Parsing [5/5]

An interesting, but impractical, method is **Valiant's Algorithm** [L.G. Valiant 1975], which does CYK using matrix multiplication. So it can benefit from fast matrix-multiplication algorithms.

Using Strassen's matrix-multiplication algorithm [V. Strassen 1969], Valiant's Algorithm parses any CFL in about $\Theta(n^{2.807355})$.

Using the Alman-Duan-Williams-Xu-Xu-Zhu matrix-multiplication algorithm [J. Alman, R. Duan, V.V. Williams, Y. Xu, Z. Xu, & R. Zhou 2024], Valiant's Algorithm parses any CFL in about $\Theta(n^{2.371339})$. But this method only achieves speed-ups for *extremely* large matrices; it is slow for realistic input.

Remember:

Practical lexers and parsers run in linear time.

Parsing Wrap-Up
Parsing in Practice [1/7]

Q. Are the methods we have covered useful in production code?

A. Lua would not be my first choice for an implementation PL. But other than that, yes.

State-Machine lexers, Recursive-Descent parsers, and automatically generated Shift-Reduce parsers are all heavily used. So, while we have by no means looked at all known parsing methods, what we have covered will at least give you the flavor of the methods that are used in production code.

Now we take a brief look at the broader world of parsing in practice.

Parsing Wrap-Up
Parsing in Practice [2/7]

The syntax of Lua is specified with a CFG. The standard Lua implementation uses a hand-coded Recursive-Descent parser.

However, this parser does not construct an AST. Instead, it emits bytecode directly. Essentially, the parser *is* the compiler. Such a parser is called a **shotgun parser**.

A compiler based on a shotgun parser will tend to run fast. But shotgun parsing is rare, because it has a number of downsides:

- Modularity and separation of concerns are reduced. Shotgun parsers are difficult to debug and maintain.
- Processing steps that use an intermediate representation, like static typing and code optimization, are difficult or impossible to do.

Because of the second point, shotgun parsers tend to produce slow code, and they are impractical for most statically typed PLs.

Parsing in Practice [3/7]

Various alternatives to CFGs have been proposed. One of the more successful ideas is the **parsing expression grammar** (**PEG**) [B. Ford 2004].

A PEG looks very similar to a CFG. It has various additional options available. A very important difference is that, with a PEG, we always form a derivation using the *first* production that works.

In a CFG, replacing "$A \rightarrow B \mid C$" with "$A \rightarrow C \mid B$" changes nothing. But in a PEG, this replacement may result in different parse trees, or even a different language being generated.

Furthermore, with a PEG, there is never more than one parse tree. PEGs have no issues with ambiguity.

The syntax of the Python PL is currently specified with a PEG. The primary implementation, **CPython**, uses a Recursive-Descent parser that is generated automatically from this PEG.

Parsing Wrap-Up
Parsing in Practice [4/7]

Originally, the C++ parser in GCC (a.k.a. g++) was automatically generated by GNU Bison using the LALR method, with additional manual editing of the generated code. However, since 2004, GCC has parsed C++ with a hand-coded Recursive-Descent parser.

Clang also parses C++ in this way.

Historically, the world of parsing has been dominated by automatically generated bottom-up parsers, primarily LALR parsers. I would imagine that it still is. For example, Ruby and PHP use such parsers. However, other parsing methods are increasingly used.

Parsing in Practice [5/7]

Another method of interest that is actually used in practice is the **Generalized LR** (**GLR**) parser mentioned a few slides back.

A GLR parser uses a variant of the Shift-Reduce idea, but it allows for grammars that are not LR(k). Roughly speaking, it does this by allowing for multiple possible actions for a particular state and symbol, and it tries all of them.

As noted earlier, GLR parsing is generally $\Theta(n^3)$. But it is much faster for *some* grammars. As a result, GLR is considered to be a parsing method that is *sometimes* practical—but only for those grammars for which it is efficient.

Also, GLR easily handles some situations that are problematic for other parsing methods. For this reason use of GLR is on the rise.

Parsing in Practice [6/7]

Producing a parser is a very practical skill.

This might seem unlikely; for example, as a member of a software-development team, you will probably not write a compiler.

But *what is parsing*?

We have defined it: *parsing* is determining whether input is syntactically correct and, if so, finding its structure.

However, there is another way of looking at it:

> ***Parsing* is making sense of input.**

And that is something that computer programs need to do *a lot*.

Parsing Wrap-Up
Parsing in Practice [7/7]

Lastly, writing a parser is generally not a terribly difficult task.

So knowing how to produce a parser can be a useful addition to your personal toolbox.

Thoughts on Assignment 4

Thoughts on Assignment 4
Introduction [1/3]

In Assignment 4 you will write a Predictive Recursive-Descent parser, in the form of Lua module `parseit`. It will be similar to module `rdparser3` (written in class), but it will involve a different grammar & AST specification.

Your parser will call your lexer from Assignment 3. That means your lexer needs to work! If you turned in a not-quite-working lexer in Assignment 3, then fix it and turn it in again as part of your submission for Assignment 4.

In Assignment 6 you will write an interpreter that takes, as input, an AST in the form your parser returns. Your lexer, parser, and interpreter together will form an implementation of a programming language called **Tamandua**.

Thoughts on Assignment 4
Introduction [2/3]

Here again is a sample Tamandua program.

```
# fibo (param in variable n)
# Return Fibonacci number F(n).
func fibo() {
    currfib = 0;
    nextfib = 1;
    i = 0;  # Loop counter
    while (i < n) {
        tmp = currfib + nextfib;
        currfib = nextfib;
        nextfib = tmp;
        ++i;
    }
    return currfib;
}
```

```
# Main program
# Print some Fibonacci numbers
how_many_to_print = 20;

println("Fibonacci Numbers");

j = 0;  # Loop counter
while (j < how_many_to_print) {
    n = j;  # Set param for fibo
    ff = fibo();
    println("F(", j, ") = ", ff);
    ++j;
}
println();
```

Thoughts on Assignment 4
Introduction [3/3]

Here is a grammar specifying the syntax of Tamandua.

1. *program* → { *statement* }
2. *statement* → `';'`
3. | (`'print'` | `'println'`) `'('` [*print_arg* { `','` *print_arg* }] `')'` `';'`
4. | `'return'` *expr* `';'`
5. | (`'++'` | `'--'`) ID [`'['` *expr* `']'`] `';'`
6. | ID (`'('` `')'` | [`'['` *expr* `']'`] `'='` *expr*) `';'`
7. | `'func'` ID `'('` `')'` `'{'` *program* `'}'`
8. | `'if'` `'('` *expr* `')'` `'{'` *program* `'}'` { `'elsif'` *expr* `'{'` *program* `'}'` } [`'else'` `'{'` *program* `'}'`]
9. | `'while'` `'('` *expr* `')'` `'{'` *program* `'}'`
10. *print_arg* → STRLIT
11. | `'chr'` `'('` *expr* `')'`
12. | *expr*
13. *expr* → *compare_expr* { (`'&&'` | `'||'`) *compare_expr* }
14. *compare_expr* → *arith_expr* { (`'=='` | `'!='` | `'<'` | `'<='` | `'>'` | `'>='`) *arith_expr* }
15. *arith_expr* → *term* { (`'+'` | `'-'`) *term* }
16. *term* → *factor* { (`'*'` | `'/'` | `'%'`) *factor* }
17. *factor* → NUMLIT
18. | `'('` *expr* `')'`
19. | (`'+'` | `'-'` | `'!'`) *factor*
20. | `'readint'` `'('` `')'`
21. | `'rnd'` `'('` *expr* `')'`
22. | ID [`'('` `')'` | `'['` *expr* `']'`]

Thoughts on Assignment 4
Mistakes to Watch For [1/2]

The most common mistake I made when writing module `parseit` was forgetting to declare a variable as `local`.

I recommend beginning each parsing function with a `local` declaration that includes all local variables used in the function.

```
local good, ast, saveop, newast
```

My second most common mistake was to return only one value from a parsing function.

In the `parseit` module, a parsing function will always return two values: Boolean & AST.

- If the Boolean is `true`, then the AST must be in the proper form.
- If the Boolean is `false`, then the AST can be anything (it might as well be `nil`).

Thoughts on Assignment 4
Mistakes to Watch For [2/2]

A mistake I have seen many students make on an assignment of this kind involves failing to *trust* their functions in some way.

Remember:

If you have a function that does something, and you need to do that thing, then call the function.

So:

- If you need something parsed, then call the appropriate parsing function, and trust that function to do it right.
- Do not try to do a function's work for it.
- If something has already been written, then you do not need to write it again.

Thoughts on Assignment 4
Starting Out

There are three files in the Git repository that you may find helpful:

- `parseit.lua`. This is an incomplete version of my solution to the assignment. It is not all there, but the code that is there is correct. Please base your work on this file.
- `rdparser3.lua`. This is the same kind of parser as the one you are to write. In particular the expression-parsing code that you need to write will be very similar to that in `rdparser3.lua`.
- `use_parseit.lua`. This is the usual "`use_`…" program. It sends input to your parser and prints the result. You can try out different inputs by editing this file.

See `parseit.lua`, `rdparser3.lua`, `use_parseit.lua`.

Unit Overview
The Haskell Programming Language

Our fourth unit: The Haskell Programming Language.

Topics

- PL feature: type system
- PL category: functional PLs
- Introduction to Haskell
- Haskell: functions
- Haskell: lists
- Haskell: flow of control
- Haskell: I/O I
- Haskell: I/O II
- Haskell: data

At the end of this unit, the Midterm Exam will be given in class.

Then we will cover The Scheme Programming Language after Spring Break.

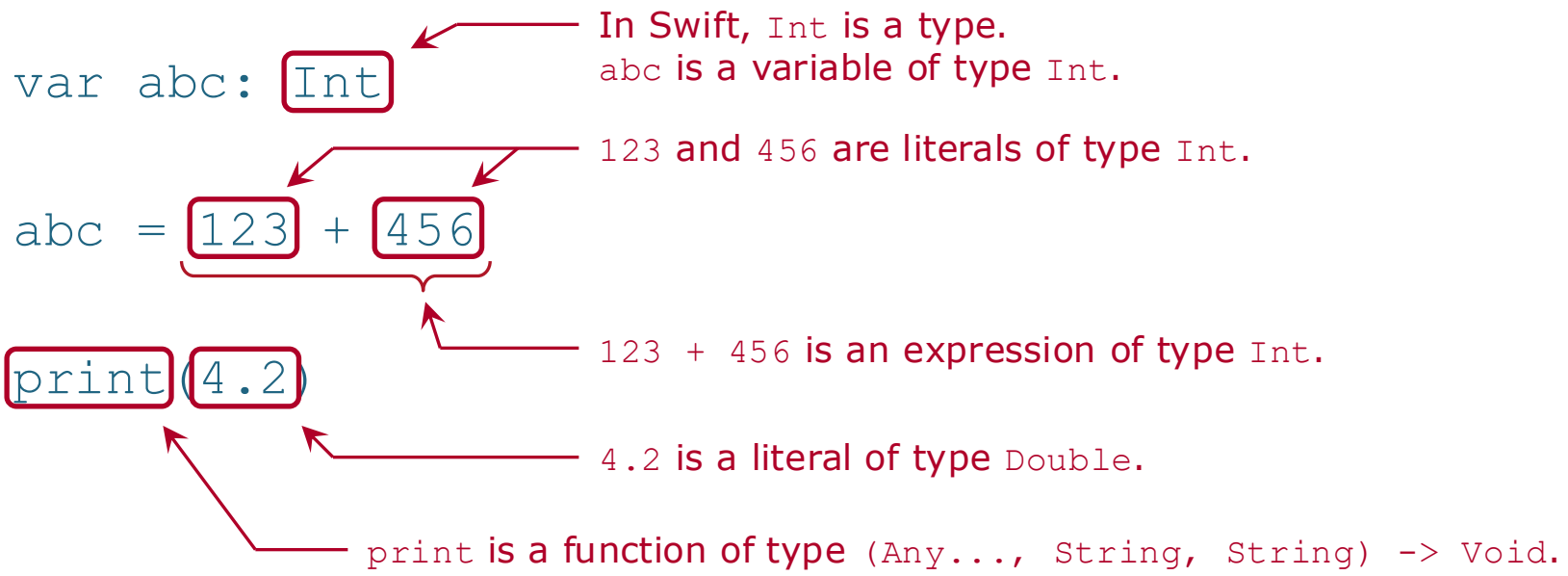
PL Feature: Type System

PL Feature: Type System
Basic Concepts [1/3]

A **type system** is a way of classifying values and/or the entities that represent them in a program, by *kind of value*, in order to prevent undesirable program states.

These slides are an incomplete summary of the reading "A Primer on Type Systems".

Each classification is a **type**.



PL Feature: Type System
Basic Concepts [2/3]

The great majority of PLs include some kind of type system.

In the past, PLs often had a fixed set of types. Many modern PLs have an **extensible** type system: one that allows programmers to define new types.

```
class Zebra {  // New type named "Zebra" (Swift or C++)
    …
```

Type checking means checking & enforcing the restrictions associated with a type system.

The various actions involved with a type system (determining types, type checking) are collectively known as **typing**.

PL Feature: Type System
Basic Concepts [3/3]

Types are used in three ways. They are used to determine:

1. Which values an entity may take on.

```
var ii: Int = "abc"  // Type error: "abc" is not Int
```

2. Which operations are legal.

```
print(3 * "abc")   // Type error: cannot multiply String
```

3. Which of multiple possible operations to perform.

```
print(123 + 456)   // + does int addition
var ss1, ss2: String
print(ss1 + ss2)   // + does string concatenation
```

PL Feature: Type System
Classifying Type Systems [1/2]

We classify type systems along three axes.

1. Overall type system: **static** or **dynamic**.
2. How types are specified: **manifest** or **implicit**.
3. How types are checked: **nominal** or **structural**.

We can also consider **type safety**.

PL Feature: Type System
Classifying Type Systems [2/2]

The following table shows how the type systems of various PLs can be classified along our first two axes.

Type Specification

Overall Type System		**Mostly Manifest**	**Mostly Implicit**
	Static	C, C++, Java	Haskell, OCaml
	Dynamic	*Not much goes here*	Python, Lua, Ruby, JavaScript, Scheme

Swift would be in **here** somewhere, depending on the coding style used.

PL Feature: Type System
Type Safety [1/3]

A PL or PL construct is **type-safe** if it forbids operations that are incorrect for the types on which they operate.

Some PLs/constructs *discourage* incorrect operations without forbidding them. We may compare their *level* of **type safety**.

In C & C++, `printf` is not type-safe. The following assumes `age` has type `int`, but does not check. It may behave oddly if `age` has a different type.

```
printf("I am %d years old.", age);  // C or C++
```

In Swift, `print` is type-safe. Below, `age` is output correctly, based on its type. This will not compile if that type cannot be output.

```
print("I am", age, "years old.")    // Swift
```

PL Feature: Type System
Type Safety [2/3]

A static type system is **sound** if it guarantees that operations that are incorrect for a type will not be performed; otherwise it is **unsound**.

Haskell has a sound type system. C and C++ have unsound type systems.

This is not a criticism! The type systems of C and C++ are intentionally unsound.

In the world of dynamic typing, there does not seem to be any standard terminology corresponding to soundness. However, we can still talk about whether a dynamic type system strictly enforces type safety.

PL Feature: Type System
Type Safety [3/3]

Two unfortunate terms are often used in discussions of type safety: **strong typing** (or **strongly typed**) and **weak typing** (or **weakly typed**). These generally have something to do with the overall level of type safety offered by a PL.

But these terms have no standard definitions. They are used in different ways by different people. (I have seen at least three definitions of "strongly typed" in common use. C is strongly typed by one of them and weakly typed by the other two.)

Therefore:

Avoid using the terms "strong" and "weak" typing, or "strongly typed" and "weakly typed".